A fully decentralized triplestore managed via the Ethereum blockchain

Damien GRAUX [a] and Sina MAHMOODI [b]

[a] *Inria, Université Côte d'Azur, CNRS, I3S, France* - `damien.graux@inria.fr`

[b] *Ethereum Foundation* - `sina.mahmoodi@ethereum.org`

Abstract. The growing web of data warrants better data management strategies. Data silos are single points of failure and they face availability problems which lead to broken links. Furthermore the dynamic nature of some datasets increases the need for a versioning scheme. In this work, we propose a novel architecture for a linked open data infrastructure, built on open decentralized technologies. IPFS is used for storage and retrieval of data, and the public Ethereum blockchain is used for naming, versioning and storing metadata of datasets. We furthermore exploit two mechanisms for maintaining a collection of relevant, high-quality datasets in a distributed manner in which participants are incentivized. The platform is shown to have a low barrier to entry and censorship-resistance. It benefits from the fault-tolerance of its underlying technologies. Furthermore, we validate the approach by implementing our solution.

Keywords. RDF store, Decentralized solution, Versioning management, Smart contracts

1. Introduction

Over the last decade, as more and more linked data in the form of RDF [7] triples were published, a set of data management practices [18] were proposed and adopted which aimed to improve integration and reuse among datasets, forming the web of data, which can be seen as a global namespace connecting individual graphs and statements. From a logical point of view, linked data is inherently decentralized. However, from a practical point of view, the actual data reside on data silos which suffer from low availability [10], leading to broken links. Furthermore, when considering dynamic datasets [19], a lack of robust versioning scheme can lead to inconsistencies when an external linked dataset is modified. But versioning datasets using HTTP has so far proven difficult [27]. Another implication of the unprecedented volume of data being published in web of data is the varying quality of datasets. Expert quality assessment [33] and curation produces the best result, but in large scale incurs high costs in terms of expert time and labor.

The contributions of this work include a novel architecture for a decentralized linked open data infrastructure, based on IPFS [4] and the public Ethereum blockchain [32]. The design includes an indexing scheme suitable for linked data, and a mechanism for retrieval of data by performing triple pattern or SPARQL queries. It further outlines how smart contracts can be employed to provide a persistent identifier for data objects stored on IPFS, to describe and version datasets, to control write access and to ensure source of provenance. A prototype of the aforementioned architecture has been implemented, and is available under an open license[1]. Moreover, on this foundation, and to further explore crowdsourcing data curation in scale, we exploit two mechanisms, first proposed by Ethereum community members [5,14], which facilitate distributed, truthful, incentivized consensus on a curated list of datasets. The mechanisms are agnostic to the domain and the actual quality metrics.

The rest of this article is divided as follows. First we provide some necessary presentation about the considered technologies in Section 2. In Section 3, we present the decentralized architecture we propose to store data; and the Ethereum smart contract solutions we set up to manage the knowledge graphs (KG) in Section 4; before presenting data curation strategies in Section 5. Then we show how data can be retrieved and report on experimental validations in Section 6. Finally we review related work and conclude in Sections 7 and 8.

2. Background

IPFS [4] is a peer-to-peer protocol for content-addressable storage and retrieval of data. It is a peer-to-peer network, with no difference between the participating nodes. It utilizes routing mechanisms to keep track of data storage, and block exchange mechanisms to facilitate the transfer of data. Every node stores IPFS objects in their local storage. These objects could be published by the node, or retrieved from other nodes and replicated locally. Objects in IPFS are comprised of immutable content-addressed data structures (such as files), that are connected with links, forming a Merkle DAG (directed acyclic graph). Addressing is done using cryptographic hashes. Content can be identified uniquely by its hash, and after retrieval, the integrity of it can be verified against the hash that was used to address it. IPFS, however, does not guarantee persistence, only permanence. A piece of content can always be referred by its hash, but it doesn't necessarily exist in the nodes of the network at all times.

IPLD[2] is a data model that aims to provide a unified address space for hash-linked data structures, such as IPFS objects, git objects, Ethereum transaction data, etc., which would allow traversing data regardless of the exact underlying protocol. The benefits of such a data model include protocol-independent resolution and cross-protocol integration, upgradability, self-descriptive data models that map to a deterministic wire encoding, backward-compatibility and format-independence. A key aspect of IPLD, is a self-describing content-addressed identifier format, called CID[3] which describes an address along with its base, version of the CID format, format of the data being addressed, hash function, hash size and finally the hash (address). This allows CID to address objects from various

[1]Our implementation is provided on Github: https://github.com/dgraux/open-knowledge
[2]https://github.com/ipld/specs
[3]https://github.com/ipld/cid

protocols. IPLD, inter alia, defines merkle-dag, merkle-links and merkle-paths. Merkle-dag is a graph, the edges of which are merkle-links. A merkle-path, is a unix-like path, that traverses within objects, and across objects by dereferencing merkle-links.

Ethereum [32] For the purposes of this study, Ethereum smart contracts can be seen as state machines, that are deployed to the network along with an initial state, and the code necessary for future state transitions, by way of invoking public functions. Upon deployment, they will be assigned an address, which can hence be used to interact with them. This interaction takes place, by crafting a transaction containing the target address, the sender, value of ether to be transferred, and if target is a contract, the input data passed to the contract.

Transactions are broadcast to the network, and so-called miners propose blocks which contain a list of the previously broadcast transactions. Every other node, upon receiving a block, runs all transactions inside, and validates the computed state, against the state put forth by the miner. Miners receive a reward in ether, the native currency of the network, for helping secure the network, and to protect against Sybil attacks [8], miners compete for proposing blocks by solving a Proof of Work [22].

As mentioned, every node in the network verifies every block, which imposes a limit on the size and frequency of blocks, which results in a limited number of slots for transactions. Users, compete for the limited slots, by sending gas (in ether) along with their transactions, which the miner earns for including the transaction in a block. Gas also acts as a deterrent for spamming the network. Miners, often employ the simple strategy of including transactions which have the most payoff.

3. A fully decentralized storage system

Our proposed architecture relies on two open technologies. First, IPFS for the actual storage and retrieval of raw data (see Section 6), and Ethereum, for tracking ownership, versioning and other metadata belonging to the KG (more details in Section 4), and later on, as will be discussed, for decentralized curation of datasets (cf. Section 5).

Permissioned, centralized triplestores often store all inserted triples in a single index. However, this is not desirable in a permissionless setting where any entity has write access to the same store, meaning, entities can even publish triples that are in conflict with others already in the triplestore. Hence, in our effort, KGs are not only conceptual, but they are actually stored in separate indices, that are managed by their corresponding publishing entity. The KGs are still connected by the URI scheme, and it is possible to do federated queries across multiple KGs. One can imagine KGs to be the counterpart of servers which contain a single dataset as opposed to multi-dataset repositories. The access control mechanism is controlled on the blockchain level (Section 4), and the P2P storage layer is agnostic to access. Due to the immutable nature of IPFS, this introduces no conflict, as each modification to an existing object results in a totally new object with a new address, regardless of who has published the modified dataset.

Each KG is indexed as a Hexastore [31] on IPFS. However, the Hexastore is not stored as a single data object, but is rather broken into smaller data nodes,

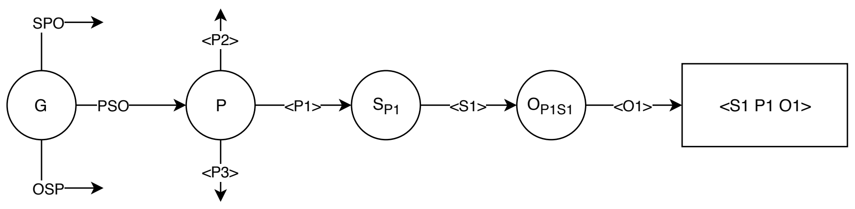

Figure 1. Example Hexastore Merkle-DAG for graph G.

which are connected via links, forming an IPLD merkle-dag. Practically, each KG has a root object, with 6 keys, namely `spo, sop, pso, pos, osp, ops` where the value of each key is a link to another object containing the triples for that subindex. Each subindex is itself a merkle-dag of depth 3. It contains the first part of the triple, with links to objects containing the second part, which in turn have links to objects containing the third part. The leaves are a simple boolean, indicating that the triple with part 1, 2 and 3 exists in the index.

As an example, consider graph G, as shown in Figure 1. The figure only displays the merkle-path to the triple `<S1 P1 O1>`, via the subindex `pso`. P is the set of all predicates in G. If `<P1>` is one of those predicates, by traversing the link for `<P1>`, we arrive at the object S_P1, which is the set of all subjects in G for which at least one triple exists with predicate `<P1>`. In a similar manner, if `<S1>` is a subject in S_P1, by traversing the link, we arrive at the object O_P1S1, which is the set of all objects in G for which triples exist with the triple pattern `<S1 P1 ?O>`. Traversing the link for `<O1>` we arrive at the leaf object `{ "exists": true }`. In Figure 1, only the path for subindex `pso` is shown. However, the same triple is indexed under the other subindices. Therefore, if G has the root hash `QmAA...AA`, the merkle-paths `QmAA...AA/pso/P1/S1/O1`, `QmAA...AA/sop/S1/O1/P1`, etc. would all be `true`.

4. A smart-contract based management system

So far we've seen the structure of indices, how KGs are stored. Each graph G is identified by the multihash of the root of its index, and updating the graph results in a completely new and unpredictable root hash. As a result, data consumers need a means for tracking the history of changes to G and consequently its root hashes, in order to be able to perform queries. In this section, two **smart contracts**, namely `Graph` and `SimpleRegistry`, will be introduced, which facilitate tracking the history of graphs and their metadata, and improving findability by naming them.

4.0.1. Graph

The `Graph` smart contract is meant to represent a single dataset, maintained by a single entity. It tracks the history of the graph, stores relevant information such as version, and points to additional metadata that the author wishes to attach to their dataset.

When creating a new KG, the publisher must publish the RDF triples on IPFS, as outlined in the previous section, and deploy an instance of the `Graph` contract, providing the root hash of the index as input. The deployed instance has a permanent address, which they can distribute to data consumers. Consumers can then query the Ethereum blockchain to fetch the state of the aforementioned contract instance, find the current root hash which they can use to perform queries. To update the KG, they update the index on IPFS, and make a transaction to the contract, providing the new hash as input. Consumers who are subscribing to events emitted by Ethereum, will be informed of the new root hash. The smart contract holds the following state fields:

Listing 1: State of the Graph contract in Solidity. For brevity, the rest of the contract has been omitted.

```
contract Graph { address public owner; uint public version;
   bytes32 public id; bytes32 public root;
   bytes32 public metadata; bytes32 public license; }
```

Ownership – In listing 1, `owner` refers to the Ethereum account who deployed the smart contract. From then on, only `owner` is able to modify the state of c, but ownership can easily be transferred to other accounts by submitting a transaction, invoking the specific `setOwner` method.
History – The field `root` is an IPFS hash which points to the root of the knowledge graph's hexastore index in IPFS. When G is updated, `owner` sends a transaction to c, updating `root`. This removes the need for a side-channel to announce new versions of g, and the need for maintaining a list of previous `roots`, as Ethereum full-archive nodes store all of the previous states by default. Furthermore, versions of G are automatically tagged by an auto-increment `version` field, which can be used to query specific versions of G without referring to the full IPFS hash in SPARQL, as will be discussed in the next section.
Metadata & Attribution – The smart contract also keeps an optional `metadata` field, which is an IPFS hash. The IPFS object identified by `metadata` could contain additional information about the knowledge graph such as details about the authors, citations to other graphs or a website link.

4.0.2. Simple Registry

The `SimpleRegistry` contract (R) acts as a KG name registry, and a list for data consumers to find KGs. Without it, data consumers would have to know the address for every KG, and would have to specify that address in their queries. R allows registering graphs under a unique name, and later on request the contract address for a certain graph with its name. It's important to note that, this contract is also openly available, and an instance of it can be deployed by any party. Data producers can decide which registry they want to be a part of. `SimpleRegistry` also allows a convenience method `newGraph(bytes32 _name)` for deploying an empty `Graph` contract and thus registering a name for it in one transaction.

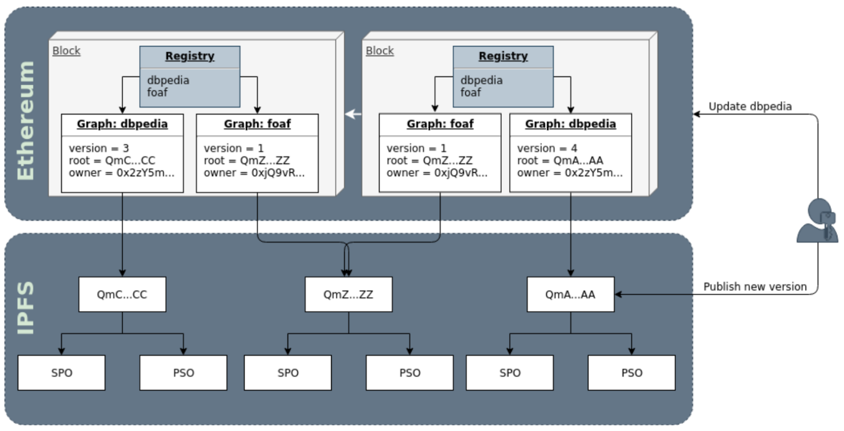

Figure 2. General Architecture.

4.0.3. The General Architecture

As presented in Figure 2, the general architecture of our solution is twofold. The bottom layer is made of the IPFS and the upper one is in the Ethereum blockchain. The various structural levels of the managing issues represented in the Ethereum are displayed in Figure 2: inside a specific block, the registry points of the associated graphs it includes and thereby could provide us with the version number and all the necessary piece of information. For example, when users want to update data, they have to publish a new version in IPFS and then update the Ethereum via the smart contracts accordingly.

5. Scalable Curation

With a growing number of datasets, it becomes increasingly costlier for consumers to find knowledge graphs suitable for their purposes, and upon finding graphs, for them to gauge their quality. This highlights the need for scalable curation mechanisms, which we will try to address in the following chapter. Free participation and censorship-resistance in our architecture has two sides. On the one hand, these characteristics ease the publication of useful and high-quality data for everyone. On the other hand, they make the infrastructure prone to being flooded with low-quality and less relevant data.

Any entity can easily create as many KGs as desired. The gas costs act as a deterrent for spamming the network. Even so, the number of legitimate graphs could potentially increase to be high enough, as to make the cost of finding suitable graphs among them non-trivial, assuming there exists a channel from which consumers can find the address of all graphs. Moreover, because `Graph` contracts have a unique and persistent identifier, namely the address of the contract, it's possible to create public lists (or collections) of graphs that are relevant for a given purpose or satisfying certain quality requirements, which consumers can refer to.

The goal is therefore to consider curation mechanisms, the output of which is a list of valid and relevant KGs for data consumers. In the following, two mechanisms will be discussed.

5.1. Adjudication via Prediction Markets

As seen previously, our architecture only stores the root hash of the index stored on IPFS, and not the index itself in the Ethereum smart contract. The main reason behind that, is transaction costs that are incurred due to storage.

As a consequence, the smart contract has no way of verifying whether the hash h actually points to a valid knowledge graph stored on IPFS. However, using Merkle proofs, or a zk-SNARK[25] proof, it is possible to prove to the smart contract, that a valid graph index would result in h as root of the index. Although verifying this proof is much cheaper than sending or storing the whole graph index, the transaction cost is still high enough to make it infeasible to do for every graph update. Novertheless, if we have the expensive method M for verifying the validity of a graph on-chain (e.g. a zk-SNARK verifier), by utilizing prediction markets, we can still check a larger number of graphs for validity, with only a smaller subset of them needing to revert to M for verification [5].

Any entity e_1 could claim that a given graph g is invalid by creating a bet of size x in the prediction market. If e_2 doesn't agree with g being invalid, they would put a bet of size y on the opposite side. If, after a pre-specified period has passed, no other entity has challenged the bet, g would hence be considered as invalid. Otherwise, verifying via M the winning side is determined. Each entity in the winning side is rewarded proportional to their bet, a part of the bets of the losing side. The process can be further optimized to deter incorrect betting and volume manipulation, by having the amount won to be only 75% of the amount lost. The other 25%, could for example be distributed to producers of valid graphs.

The rationale here is that, verifying the validity of a graph is much cheaper done off-chain, than on-chain. Therefore, users would be incentivized to "fish" invalid graphs. They are disincentivized to bet against a valid graph, because others can challenge the bet in the market, and an on-chain verification would result in the loss of their bet.

5.2. Token-Curated Registry

Token-curated registry (TCR) [14] is a mechanism, in which rational actors are incentivized to maintain a decentrally-curated list. As the name suggests, TCRs rely on a native token, which has a value relative to another base currency (fiat currencies, such as EUR). Apart from consumers of the curated list, which desire a high-quality list of KGs, other actors require tokens to interact with the TCR.
Actors – Actors of a TCR include candidates, voters and challengers. A candidate, is an actor, who wishes to add a graph to the list, and stakes N tokens along with the application. A challenger, is an actor, who believes the item that a candidate proposed, does not belong in the list, and is willing to stake N tokens to challenge the application. When a challenge occurs, a voting period starts, during which, token holders can cast a vote, either for or against the item in question.

Votes are weighted proportional to the number of tokens the token holder specifies. The tokens would not be spent during a vote. After the voting period comes to an end, the side with most token-weighted votes wins, and depending on the outcome, either the candidate or the challenger loses a portion of their stake, and this portion is split among the winners, in proportion to the number of tokens they participated with.

Rewarding honest behaviour – The rationale behind TCRs is that, rational voters seeking to increase their long-term profit, would vote to accept items that have a higher quality, which increases usefulness among consumers, resulting in more demand among candidates to be listed, increasing the value of the native token with respect to the base currency. This complements their short-term benefit of being rewarded with more tokens, if they vote for high-quality items.

Disincentives – The risk associated with losing the stake disincentivizes candidates to apply for a graph, they consider either of low quality or invalid. At the same time, challenging a high quality graph also comes with the risk of losing a portion of challenger's stake. If this was not the case, participants would have been incentivized to challenge every application, effectively requiring a vote on every application, and thereby reducing the efficiency of the mechanism.

Vote-splitting – The aforementioned specification failed to address the "nothing at-stake" problem for voters, or in particular, the "vote-splitting" issue, in which, a rational strategy for voters could be to split their tokens in half, and vote for both side, thereby earning revenue regardless of the outcome of the vote, and without putting in any effort. TCR v1.1[4] addresses this issue, by slashing a portion of the minority bloc's tokens, and adding it to the rewards of the majority's bloc.

Commit-reveal voting – Due to Ethereum transactions being public, during a voting period, voters can see the current tally, and vote with the majority, without inspecting the item in question. This can be prevented, by splitting the voting period into two phases: first, all voters make a cryptographic commitment to a vote, after the commit period has come to an end, everyone must reveal their vote, by submitting the secret used to make the commitment. Consequentially, the tally of the votes is unknown by everyone other than the voter until the end of the commit period. This effectively prevents voters from basing their decisions on how others are voting.

Listing item status – Graphs are added to the list, either if they face no challenge after application, or if they are challenged, and voters vote for inclusion of the graph. The stake, which is a requirement of applying to the TCR, remains locked while the item is in the list. The candidate, can, at any moment withdraw the stake, and thereby removing the item from the list. Furthermore, even after a graph has been listed, it can be challenged and therefore removed from the list. This is inevitable, because an append-only list, could grow large enough to lose its usefulness, and as such, when higher quality graphs are added to the list, lower quality graphs can be challenged and removed, in order to maintain a limited number of slots in the list.

[4] https://medium.com/@ilovebagels/token-curated-registries-1-1-2-0-tcrs-new-theory-and-dev-updates-34c9f079f33d

6. Retrieval

As seen in the previous sections, triples of a knowledge graph are stored in the form of a merkle-dag on IPFS. Merkle-paths allow querying triple patterns, but not other features of an advanced query language such as SPARQL [16]. It is however possible to perform a subset of all SPARQL constructs, by combining the results of several triple pattern searches. First, we will demonstrate how a simple triple pattern search can be performed, and then discuss how full SPARQL queries, either on single graphs or a federation thereof, can be executed by using triple patterns as building blocks.

6.1. Triple Patterns

A triple pattern, is a triple where any of the parts can be a variable instead of a concrete value. In the simplest case, it is possible to query the existence of a triple, that has no variable, in the KG. In this case, the merkle-path for the triple `<a b c>` would look like `QmAA...AA/spo/a/b/c` which returns `true` if the triple exists, and throws an error otherwise.

Given a graph G which has root hash G_h and a triple pattern T, the algorithm for constructing the corresponding merkle-path P and retrieving the values at this path is given below:

1. Initialize P to G_h
2. Parse T to get list of fixed and variable parts
3. Compute best subindex: bring fixed parts first, then append variable ones
4. Add subindex to P
5. Append values for fixed parts to merkle-path, separated by a "/"
6. Fetch result (R) of P from IPFS
7. If result is nonempty, construct triples by adding the values for fixed parts to the results which were returned for the variable parts, and return them.

As an example, running the algorithm over a KG which contains `[<a b c>, <f b c>]`, with `T = <?s b c>`, implies to use a *pos* index and will result in `P = QmAA...AA/pos/b/c` and `R = [a,f]`, and the algorithm will return `[<a b c>, <f b c>]`.

6.2. SPARQL Queries

Although querying triple patterns and compositions thereof would suffice for some applications, it falls short for others. In order to allow SPARQL queries, we build on the Linked Data Fragments framework[29] by implementing the Triple Patterns Fragments interface. By doing so, the TPF client decomposes a SPARQL query into triple patterns, retrieves the corresponding responses, and computes the final SPARQL response therefrom.

In implementing the TPF interface, some specific points had to be taken into account. TPF has been design with REST APIs in mind. In our architecture, the client and the implementation of the TPF interface reside on the same node, and they communicate via faster intra-process means. In addition, the original TPF interface implementation runs on a server and fetches data from a local database, whereas with our solution, triples are stored across peers, and in case the required triples are not replicated locally, each triple pattern query is automatically translated into requests to fetch triples from other peers. Finally, pagination and control functions such as `nextPage`, are a requirement of the TPF interface; currently pagination is done after fetching all of the triples matching a pattern.

Federated SPARQL queries –i.e. queries requesting several graphs at once– are performed in a similar manner, by utilizing the TPF interface. In a nutshell, the query is first split according to the various implied graphs to obtain distinct Linked Data Fragments; then we process as described above; before joining together the results. However, whereas TPF originally requests the result of triple patterns from servers via HTTP, in our architecture, all triple pattern queries are done simply over different graphs which exist on the same P2P filesystem following the concerned hash addresses.

6.3. Complete Evaluation Process

More generally, as already presented in Figure 2, the solution has two layers: one on top based in Ethereum which offers managing features and the bottom one on IPFS where data is actually stored. As a consequence, during the query phase, the system takes into account this specificity. We present in Figure 3 the details of the query process around the information retrieval. Indeed, the first step requires to look for graph addresses in the Registry, then to obtain the correct root hashes from each Graph (taking e.g. into account the version number). Once these information are obtained, we are done with the Ethereum side and can decompose the query in order to get the triple pattern from the IPFS. Once they are retrieved, the answer can be constructed and the complete process to evaluate a SPARQL query is done.

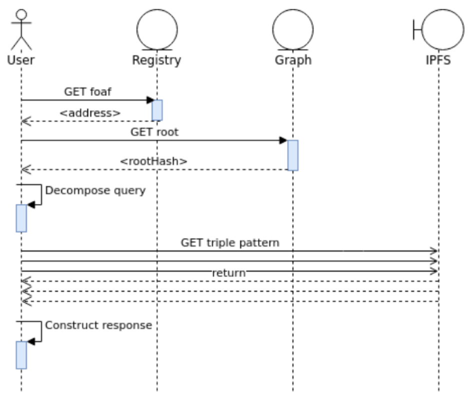

Figure 3.: Evaluation Process.

6.4. Experimental Validation

In this section, the results of a benchmark performed on the prototype implementation is presented. To verify the correct functionality of the architecture and its implementation, the source code also includes a test suite. Moreover, we generated a WatDiv [2] dataset with scale 1 which contains 107 665 triples, stored the dataset on our system and performed the 20 queries provided in the WatDiv (v0.6) packaging, comprising of linear queries (L), star queries (S), snowflake-shaped queries (F) and complex queries (C). The benchmark was executed on a computer with Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz, SSDSA2BW16 disk and 8

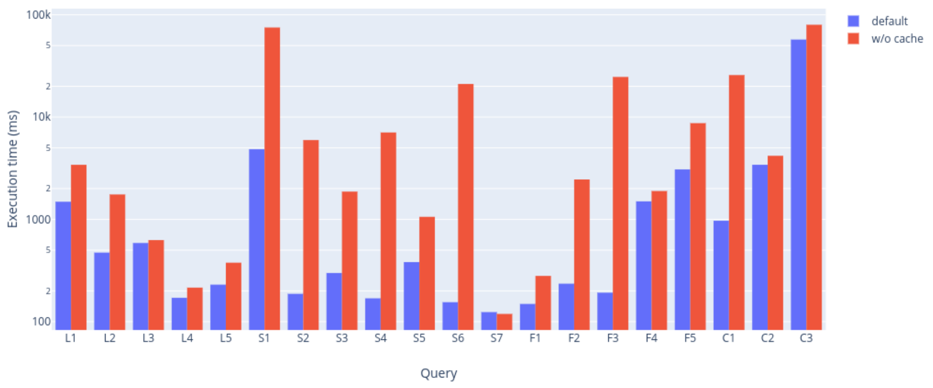

Figure 4. Query execution times.

GB of memory, running a Linux kernel (v4.19.1). We set up virtual nodes for IPFS to test our architecture. The goal was more to validate the feasability of the approach rather than benchmarking it. By default the SDK maintains a cache of results fetched from IPFS. To measure query execution times, each query has been executed 5 times with a warm cache, and 5 times with a cold cache. Results of the queries have been compared for correctness against the ARQ engine[5].

Loading Phase – Storing the dataset comprises of two main phases, constructing the index tree locally and storing the tree on IPFS from leaves to the root. Constructing the tree locally took 1 503 ms, and storing it on IPFS took 18.52 minutes and translated into 318 972 IPFS `PUT` requests.

Query Execution – Figure 4 displays execution times measured for the aforementioned queries using a logarithmic scale. The difference between *default* and *w/o cache* traces is in caching the results of IPFS `get` requests in the engine. Table 1 outlines measurements of metrics during each query, providing additional insight into factors potentially influencing the execution times. *Triple patterns* denotes the number of triple patterns each SPARQL query is decomposed into by Triple Pattern Fragments client, *IPFS gets* is the number of `GET` requests to IPFS, *Repeated paths* is the number of paths that had been requested from IPFS during the same query and *returned triples* denotes the total number of triples that have been returned from IPFS to construct the final SPARQL result.

Our implementation can be seen as a global linked open data repository which facilitates storing KGs and retrieving triples from either single graphs or a multitude of them. In particular, the metrics shown in Table 1 point to the number of decomposed triple patterns and IPFS requests as a potential factor that correlates with execution time. The *repeated paths* metric reemphasizes the benefits of a cache for intermediate results retrieved from IPFS.

[5]https://jena.apache.org/documentation/query/index.html

Query	Triple patterns	IPFS gets	Repeated paths	Returned triples
L1	131	403	79	1897
L2	26	225	3	387
L3	27	253	1	1107
L4	11	34	1	39
L5	13	296	45	297
S1	375	6849	152	8357
S2	13	990	1	1205
S3	14	305	10	664
S4	9	725	1	752
S5	16	220	1	242
S6	6	1405	3	1510
S7	3	1424	13	1499
F1	11	1743	10	2201
F2	27	2135	104	2235
F3	9	2212	1	3552
F4	271	5269	3748	6565
F5	363	68196	62629	72297
C1	51	8343	6057	10120
C2	183	17895	14906	36216
C3	3672	6851	3148	53821

Table 1. Performance metrics for WatDiv queries.

7. Related Work

There has been extensive research on centralized RDF data storage and retrieval. A survey of such storage and query processing schemes has been done by Faye et al. [11], in which triplestores are categorized based on multiple factors. These factors include native vs non-native and in-memory vs disk-based storage solutions. Non-native solutions for example are triplestores that use an existing data store, such as relational databases. Hexastore [31] stores six indices, enabling efficient lookup of triple patterns for each parts of the triple, including subject, predicate and object. This gain in performance comes at a cost in storage. When it comes to querying data from remote servers, Verborgh et al. argue that there's a spectrum between data dumps and SPARQL endpoints, and that there's a trade-off along the spectrum between factors including performance, cost, cache reuse, bandwidth, etc. for servers and clients. They propose Linked Data Fragments [29] which lies somewhere in the middle of the spectrum. In this design, clients turn a SPARQL query into a series of triple pattern requests that servers respond to, lowering load servers, decreasing bandwidth,. . . Centralized data repositories can process queries efficiently, but they are single points of failure and they have limited scalability and availability. In this study we adopt core ideas from Hexastore and LDF and apply them to the P2P network setting.

Content distribution over P2P networks has been an area of active research during the last two decades. Motivations over the client-server architecture include scalability, fault-tolerance, availability, self-organization and symmetry of nodes [17]. Androutsellis-Theotokis et al. classify P2P technologies [3] in the context of content distribution into applications and infrastructure. P2P applications themselves are classified into file exchange applications, such as Napster [23], which facilitate one-off file exchange between peers, and content publishing and storage applications, such as Publius [30], which are distributed storage schemes in which users can store, publish and distribute content securely and persistently. Technologies targeted for routing between peers and locating content have been classified under P2P infrastructure, and include inter alia Chord [28] or CAN [24]. P2P networks also differ in their degree of centralization. Some, like Napster, rely on a central server which holds metadata crucial for routing and locating content, limiting scalability, fault-tolerance and censorship-resistance, but offering efficient lookups. Lua et al. review overlay network structures, comparing *structured* and *unstructured* networks [21].

Unstructured P2P networks have been employed in protocols such as Bibster [15] and [34] to store RDF data and process queries. They use semantic similarity measures to form semi-localized clusters and to propagate queries to peers who are most likely to contain relevant data for. These protocols offer higher fault tolerance, but limited guarantees for retrieving query results even the underlying data exists in the network due to their propagation mechanisms.

Filiali et al. has performed a comprehensive survey [12] of RDF storage and retrieval over structured P2P networks. To index the triples, most protocols rely on variants of hash-indexing, e.g. RDFPeers [6], or semantic indexing, e.g. GridVine [1]. Two general strategies have been observed by Filiali et al., either retrieving all relevant triples from other peers and evaluating the result of the query locally, or propagating the query and partial results through the network, as in QC and SBV [20]. Unlike the aforementioned protocols, in this study we don't design a custom P2P network specifically built for RDF data storage, but use the live global IPFS filesystem [4], which is simultaneously being used for other purposes. Triples are indexed as a Hexastore. To process queries, all relevant triples are fetched, and result is evaluated using the Triple Pattern Fragments [29] framework.

Sicilia et al. [26] explore publishing datasets on IPFS, either by storing the whole graph as a single object or by storing each dereferenceable entity as an object. Furthermore they propose using IPNS to refer to the most recent dataset version. In this study, versioning is handled by a smart contract on Ethereum.

English et al. [9] explore both utilizing public blockchains for the semantic web, improving on the current URI schemes, storing values on the Bitcoin network, and creating ontologies for representing blockchain concepts. We share the idea that blockchains and web of data are complementary, and use the Ethereum blockchain to store metadata, and perform curation for KGs.

8. Conclusion & Future Work

In this article, we proposed a novel architecture for a fully decentralized linked data infrastructure, which has a very low barrier to entry, is censorship-resistant and benefits from fault-tolerance properties of its underlying open technologies. Due to immutability of each version of a dataset, consumers can cache the data objects they interact with, and perform queries even while offline. By replicating these data objects, they are at the same time contributing to the availability of those datasets. In addition, we explored two mechanisms which allow a community to come to consensus over a collection of datasets which they find relevant or high-quality in a distributed manner, by utilizing smart contracts to align the incentives of participants. IPFS replicates a document on every node that interacts with it. Therefore, more popular KGs are expected to be highly replicated. However, IPFS doesn't guarantee persistence. If the node that published a document goes offline, and there's no other replica, that document won't be accessible until the node comes back online. This might lower accessibility for KGs that have less demand. Future works can improve on this by incentivizing nodes to replicate pieces of data and asking them to provide *proof-of-replication* [13].

References

[1] Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: Gridvine: Building internet-scale semantic overlay networks. In: ISWC. pp. 107–121. Springer (2004)

[2] Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC. pp. 197–212. Springer (2014)

[3] Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM computing surveys (CSUR) **36**(4), 335–371 (2004)

[4] Benet, J.: Ipfs-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561 (2014)

[5] Buterin, V.: Scaling adjudication with prediction markets. https://ethresear.ch/t/list-of-primitives-useful-for-using-cryptoeconomics-driven-internet-social-media-applications/3198 (2018), accessed: April 2021

[6] Cai, M., Frank, M.: Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. In: Proceedings of the 13th international conference on World Wide Web. pp. 650–657. ACM (2004)

[7] Consortium, W.W.W., et al.: Rdf 1.1 concepts and abstract syntax (2014)

[8] Douceur, J.R.: The sybil attack. In: International workshop on peer-to-peer systems. pp. 251–260. Springer (2002)

[9] English, M., Auer, S., Domingue, J.: Blockchain technologies & the semantic web: a framework for symbiotic development. In: Computer Science Conference for University of Bonn Students. pp. 47–61 (2016)

[10] Ermilov, I., Martin, M., Lehmann, J., Auer, S.: Linked open data statistics: Collection and exploitation. In: International Conference on Knowledge Engineering and the Semantic Web. pp. 242–249. Springer (2013)

[11] Faye, D.C., Curé, O., Blin, G.: A survey of rdf storage approaches. Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées **15**, 11–35 (2012)

[12] Filali, I., Bongiovanni, F., Huet, F., Baude, F.: A survey of structured p2p systems for rdf data storage and retrieval. In: Transactions on large-scale data-and knowledge-centered systems III, pp. 20–55. Springer (2011)

[13] Fisch, B.: Poreps: Proofs of space on useful data. Cryptology ePrint Archive, Report 2018/678 (2018), https://eprint.iacr.org/2018/678

[14] Goldin, M.: Token-curated registries 1.0. https://docs.google.com/document/d/1BWWC__-Kmso9b7yCI_R7ysoGFIT9D_sfjH3axQsmB6E (2018), accessed: April 2021

[15] Haase, P., Broekstra, J., Ehrig, M., Menken, M., Mika, P., Olko, M., Plechawski, M., Pyszlak, P., Schnizler, B., Siebes, R., et al.: Bibster–a semantics-based bibliographic peer-to-peer system. In: ISWC. pp. 122–136. Springer (2004)

[16] Harris, S., Seaborne, A., Prud'hommeaux, E.: Sparql 1.1 query language. W3C recommendation **21**(10) (2013)

[17] Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., Campbell, R.: A survey of peer-to-peer storage techniques for distributed file systems. In: ITCC. vol. 2, pp. 205–213. IEEE (2005)

[18] Heath, T., Bizer, C.: Linked data: Evolving the web into a global data space. Synthesis lectures on the semantic web: theory and technology **1**(1), 1–136 (2011)

[19] Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing linked data dynamics. In: ESWC. pp. 213–227. Springer (2013)

[20] Liarou, E., Idreos, S., Koubarakis, M.: Evaluating conjunctive triple pattern queries over large structured overlay networks. In: International Semantic Web Conference. pp. 399–413. Springer (2006)

[21] Lua, E.K., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A survey and comparison of peer-to-peer overlay network schemes. Surveys & Tutorials **7**(2), 72–93 (2005)

[22] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)

[23] Napster, L.: Napster. http://www.napster.com (2001), accessed: April 2021

[24] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network, vol. 31. ACM (2001)

[25] Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy (SP). pp. 459–474. IEEE (2014)

[26] Sicilia, M.A., Sánchez-Alonso, S., García-Barriocanal, E.: Sharing linked open data over peer-to-peer distributed file systems: the case of ipfs. In: Research Conference on Metadata and Semantics Research. pp. 3–14. Springer (2016)

[27] Van de Sompel, H., Sanderson, R., Nelson, M.L., Balakireva, L.L., Shankar, H., Ainsworth, S.: An http-based versioning mechanism for linked data. arXiv preprint arXiv:1003.3661 (2010)

[28] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review **31**(4), 149–160 (2001)

[29] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web. Web Semantics: Science, Services and Agents on the World Wide Web **37**, 184–206 (2016)

[30] Waldman, M., Rubin, A.D., Cranor, L.F.: Publius: A robust, tamper-evident censorship-resistant web publishing system. In: USENIX Security Symp. (2000)

[31] Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment **1**(1), 1008–1019 (2008)

[32] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)

[33] Zaveri, A., Rula, A., Maurino, A., Pietrobon, R., Lehmann, J., Auer, S.: Quality assessment for linked data: A survey. Semantic Web **7**(1), 63–93 (2016)

[34] Zhou, J., Hall, W., De Roure, D.: Building a distributed infrastructure for scalable triple stores. Journal of Computer Science and Technology **24**(3), 447–462 (2009)